VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Implementace relačního spojení s přihlédnutím k L2 cache

Relational Cache Aware Join

2015

Tomáš Mocek

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Tomáš Mocek**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Implementace relačního spojení s přihlédnutím k L2 cache
Relational Cache Aware Join

Zásady pro vypracování:

Relační spojení je jedna z nejdůležitějších operací v relačních databázích. Efektivita této operace výrazně ovlivňuje efektivitu celého dotazu, kterého je spojení součástí. Mezi základní implementace operace spojení patří sort-merge, nested loop a hash-join. Nicméně v posledních letech se implementace těchto algoritmů výrazně upravila s ohledem na paměťovou architekturu počítačů. Cílem této práce je implementovat radix-cluster hash-join algoritmus a porovnat jej s GRACE hash-join algoritmem.

Práce bude mít následující části:
1. Seznámení se s algoritmy pro operaci spojení.
2. Seznámení se s variantami zohledňující L2 cache počítače.
3. Implementace radix-cluster hash-join algoritmu a GRACE hash-join algoritmu.
4. Porovnání těchto implementací.

Seznam doporučené odborné literatury:

[1] Kitsuregawa, Masaru, Hidehiko Tanaka, and Tohru Moto-Oka. "Application of hash to data base machine and its architecture." New Generation Computing 1.1 (1983): 63-74.
[2] Manegold, Stefan, Peter Boncz, and Martin Kersten. "Optimizing main-memory join on modern hardware." Knowledge and Data Engineering, IEEE Transactions on 14.4 (2002): 709-730.
[3] Chen, Shimin, et al. "Improving hash join performance through prefetching." ACM Transactions on Database Systems (TODS) 32.3 (2007): 17.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Radim Bača, Ph.D.**

Datum zadání: 01.09.2014
Datum odevzdání: 07.05.2015

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 16.4.2015 .....

Zde bych rád poděkoval mému vedoucímu Ing. Radimu Bačovi, Ph.D. za jeho odborné vedení, důležité rady a trpělivost, díky kterým jsem byl schopen tuto práci dokončit. Dále bych rád poděkoval svým rodičům Ivě a Martinovi, za jejich nekonečnou podporu při studiu a poskytnutí výborných studijních podmínek. Nakonec bych chtěl poděkovat svým kamarádům z ročníku, Janu Homolovi a Adamu Zikmundovi, za jejich rady, Lukášovi Zátopkovi za grafické nápady, kamarádce Adéle Kloudové za její podporu a všem ostatním, se kterými jsem se během studia kamarádil.

Abstrakt

Tato práce se zabývá teoretickým popisem procesorové cache počítače, a dále pak jejím využitím pro implementaci optimalizovaných algoritmů. Konkrétně se zaměřuje na algoritmy relačního spojení, jako jsou například nested loop a hash-join. Hlavní část práce bude pojednávat o algoritmech GRACE hash-join a radix-cluster hash-join, které jsem implementoval v jazyce C++, s důrazem na využití procesorové cache. Také zde budou zmíněny nástroje, které mohou pomoci při optimalizaci algoritmů pro cache. V závěru pak budou tyto algoritmy mezi sebou porovnány, a také bude provedeno srovnání s neoptimalizovaným algoritmem hash-join.

Klíčová slova: cache, TLB, relační spojení, optimalizace, hash-join, GRACE hash-join, radix-cluster hash-join

Abstract

This thesis deals with theoretical description of processor's cache, as well as it's exploitation in implementing optimalized algorithms. It will focus on algorithms for relational joins. These algorithms are, for example, nested loop and hash-join. The core part of the thesis will explain GRACE hash-join algorithm and radix-cluster hash-join algorithm, which were implemented in C++ with emphasis on processor's cache exploitation. Furthermore, the tools that can be used during cache aware optimalizations of algorithms will be mentioned. In the conclusion, these algorithms will be compared to each other, as well as with unoptimized hash-join algorithm.

Keywords: cache, TLB, relational join, optimalization, hash-join, GRACE hash-join, radix-cluster hash-join

Seznam použitých zkratek a symbolů

TLB	–	Translation Lookaside Buffer
SQL	–	Structured Query Language
CPU	–	Central Processing Unit
RAM	–	Random Access Memory
GPU	–	Graphics Processing Unit
HW	–	Hardware
HDD	–	Hard-Disk Drive
SSD	–	Solid-State Drive
DNS	–	Domain Name System
P2P	–	Peer To Peer
L1	–	Level 1 cache
L2	–	Level 2 cache
L3	–	Level 3 cache
LLC	–	Last Level Cache
LRU	–	Least Recently Used
FIFO	–	First In First Out
ns	–	nanosekunda
MB	–	megabajt
kB	–	kilobajt
MMU	–	Memory Management Unit
VPN	–	Virtual Page Number
PFN	–	Page Frame Number

Obsah

Seznam tabulek

Seznam obrázků

Seznam algoritmů

1 Úvod

Za posledních několik desetiletí došlo k extrémnímu zvýšení výkonu počítačů, které ovšem nebylo rovnoměrně rozděleno mezi všechny hardwarové komponenty. Například výkon procesoru vzrostl za poslední dekády mnohonásobně více, než výkon RAM pamětí, a tím se přístup do paměti stává úzkým místem celého systému. Pro demonstraci, jednoduchý průchod databázové tabulky vykazoval zhruba stejný celkový čas na hardware z roku 1992, jako tomu bylo na hardware z roku 2000. Tato situace je způsobena tím, že na novějším zařízení stráví procesor 95% cyklů čekáním na data z paměti[1].

Těchto poznatků lze využít při implementaci algoritmů, jako jsou například algoritmy relačního spojení, jejichž efektivita může výrazně ovlivnit efektivitu celého dotazu, kterého je spojení součástí, a kterému bude tato práce věnovat největší pozornost.

V kapitole 2 budou představeny základní algoritmy pro relační spojení, které ovšem nevyužívají skutečností zmíněných výše. V kapitole 3 a 4 pak bude podrobně popsána procesorová cache a TLB, které lze využít pro zmenšení zpoždění vyvolaného pomalým přístupem do paměti, a také programátorské praktiky, jimiž lze kód pro cache optimalizovat.

V kapitole 5 a 6 se bude práce podrobně věnovat algoritmům spojení využívající právě procesorovou cache. Tyto algoritmy jsou GRACE hash-join a radix-cluster hash-join, jejichž implementace budou testovány a výsledky pak budou v kapitole 7 porovnány jak mezi sebou, tak spolu hash-join algoritmem, který tyto optimalizace nevyužívá.

V závěru pak ještě bude práce porovnána s výsledky jiné práce[1] z roku 2002.

2 Operace relační spojení

Spojení, neboli join, je operace, která kombinuje záznamy dvou různých tabulek na základě spojovací podmínky. Výsledkem spojení je opět tabulka, se kterou pak můžeme dále pracovat. ANSI standard pro SQL definuje 5 druhů spojení. Mezi ně patří INNER JOIN, LEFT OUTER JOIN, RIGHT OUTER JOIN, FULL OUTER JOIN a CROSS JOIN. Jako další typ spojení lze považovat SELF JOIN nebo SEMI JOIN, pro které ale v SQL neexistuje explicitní výraz.

```
SELECT * FROM Oddeleni o JOIN Zamestnanec z ON o.IdOddeleni = z.IdOddeleni
```

Výpis 1: Ukázkový příkaz SELECT v jazyce SQL

2.1 Základní druhy spojení

Pravděpodobně nejvíce používaný druh spojení je INNER JOIN(vnitřní spojení), který pro každý prvek z jedné relace prochází každý prvek z relace druhé. Výsledkem je pak tabulka obsahující ty záznamy obou relací, které splňují spojovací podmínku.

Dalšími, méně používanými, ale zdaleka ne méně důležitými jsou LEFT OUTER, RIGHT OUTER a FULL OUTER JOIN, které patří do kategorie vnějších spojení. Stejně jako u INNER JOINu se také vytváří nová množina dle spojovací podmínky. Rozdíl mezi nimi spočívá v rozsahu prvků, které se mohou do výsledné relace uložit.

Například při LEFT OUTER joinu se do výsledné relace po této operaci uloží narozdíl od INNER JOINu všechny záznamy první relace, a to i ty, ke kterým nebyl spojovací podmínkou nalezen žádný záznam z relace druhé. Tyto atributy z druhé relace jsou pak nahrazeny NULL hodnotami. Analogicky lze pochopit i ostatní spojení.

Mezi speciální druhy spojení lze pak považovat CROSS JOIN, SELF JOIN nebo SEMI JOIN. CROSS JOIN vytváři kartézský součin spojovaných tabulek a SELF JOIN je druh spojení, kdy je tabulka spojena sama na sebe. SEMI JOIN pak vrací pouze záznamy první relace, ke kterým by existoval záznam v relaci druhé a lze jej použít například k omezení množství přenášených dat.

Poznámka 2.1 Při práci s NULL hodnotou je pak nutné si uvědomit některá úskalí, jelikož bývá nesprávně považována za synonymum prázdné nebo nulové hodnoty. Ve skutečnosti by se ale o NULL hodnotě melo uvažovat jako o neurčité hodnotě s tím, že platí, že NULL se nerovná NULL.

2.2 Algoritmy implementující relační spojení

Existuje mnoho rozdílných implementací relačního spojení, každé se svými výhodami a nevýhodami. Mezi základní algoritmy patří *nested loop*, *sort-merge* a *hash-join*.

2.2.1 Nested loop join

Jedná se o základní algoritmus operace spojení, který jak můžeme vidět na algoritmu číslo 1, využívá dva zanořené cykly. Nejdříve se prochází první relace a pro každý její prvek se vyhledávají v druhé relaci záznamy, pro které je splněna spojovací podmínka. Asymptotická složitost algoritmu je $\Theta(|R| * |S|)$, kde $|R|$ je velikost (počet prvků) relace R a $|S|$ je počet prvků relace S. Je zřejmé, že tento algoritmus není vhodný pro relace s velmi velkými počty prvků (např. milióny, ale i řádově nižší), nýbrž s rostoucím počtem prvků se složitost kvadraticky navyšuje, a je tedy vhodný, když je jedna z relací velmi malá.

Algoritmus 1: Nested loop join

```
   input  : relace R, relace S
   output: relace Q
1  for each r v R do
2      for each s v S do
3          if r a s splňují spojovací podmínku then
4              vlož r a s do Q;
5          end
6      end
7  end
```

2.2.2 Sort-merge

Sort-merge join je dalším z algoritmů relačního spojení a popisuje jej algoritmus číslo 2. Princip spočívá v tom, že obě vstupní relace jsou nejdříve setřízeny podle spojovacího atributu. Prvky relací, které splňují spojovací podmínku, jsou poté součástí výsledné relace.

Výhodou algoritmu je, že pokud jsou obě relace setřízené, máme složitost celého algoritmu $\Theta(|R|+|S|)$. Nevýhoda je právě v požadavku, že obě relace musí být setřízené, a třídění tedy degraduje ideální složitost celého algoritmu na $\Theta((|R|log|R|) + (|S|log|S|) + (|R| + |S|))$ za předpokladu, že pro třídění použijeme *merge sort*[2].

2.2.3 Hash-join

Posledním ze základních představovaných algoritmů je hash-join. Oproti nested-loop, který se hodí na data malých velikostí, a sort-merge joinu, který se hodí na data průměrných velikostí, je hash-join vhodný na spojování tabulek s největším počtem záznamů.

Také je dobré zmínit, že tento algoritmus lze na rozdíl od předchozích použít pouze na *Equi-join*, ve kterém je spojovací podmínka založena na rovnosti hodnot specifikovaných sloupců. Průběh tohoto algoritmu popisuje algoritmus číslo 3.

Algoritmus se skládá ze dvou základních částí, *build* fáze (viz řádek číslo 1) a *probe* fáze (viz řádek číslo 8)[3].

Build fáze je přípravná metoda, která z menší relace vytvoří pomocí hashovací funkce hash tabulku. Probe fáze je pak metoda, která pro každý prvek větší relace, respektive pro spojovací atribut nalezne odpovídající pole hash tabulky, ve které se může nacházet prvek splňující spojovací podmínku.

Tyto algoritmy lze také implementovat s přihlédnutím ke cache procesoru a tím rapidně snížit jejich rychlost. V následující kapitole bude tato cache popsána a také bude zavedeno názvosloví, které je nutné ovládat pro porozumění a tvorbu optimalizovaných algoritmů.

Algoritmus 2: Sort-merge[4]

```
   input  : relace R, relace S
   output: relace Q
 1 setřiď R;
 2 setřiď S;
 3 vyber první záznam r z R;
 4 vyber první záznam s ze S;
 5 while R a S není na konci do
 6 |   if r a s splňují spojovací podmínku then
 7 |   |   vlož r a s do Q;
 8 |   |   vyber další záznam s ze S;
 9 |   else if r < s then
10 |   |   vyber další záznam r z R;
11 |   else
12 |   |   vyber další záznam s ze S;
13 |   end
14 end
```

Algoritmus 3: Hash-join

```
   input  : relace R, relace S
   output: relace Q
 1 for each r v R do                                          // Build fáze
 2 |   vypočti hash pole pro r;
 3 |   vlož r do vypočteného hash pole;
 4 |   if hash pole je plné then
 5 |   |   zvětši hash pole;
 6 |   end
 7 end
 8 for each s v S do                                          // Probe fáze
 9 |   vypočti hash pole pro s;
10 |   for each p ve vypočteném hash poli do
11 |   |   if p == s then
12 |   |   |   vlož p a s do Q;
13 |   |   end
14 |   end
15 end
```

3 Cache

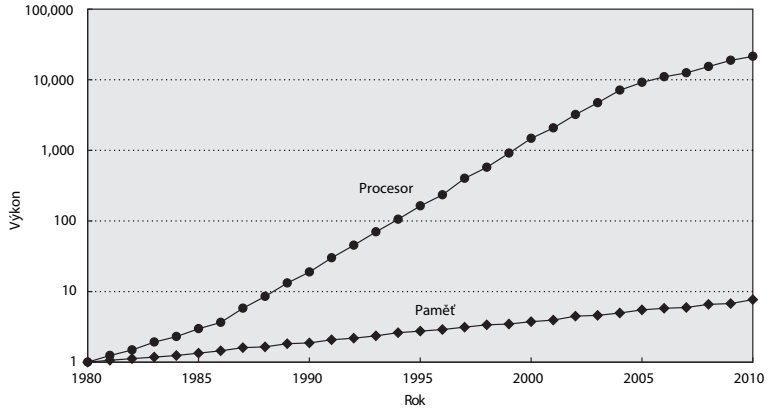
Cache (*Cache Memory*), neboli česky mezipaměť, je v informatice označení pro komponentu, která slouží k omezení úzkého místa mezi dvěma systémy s různými rychlostmi. Účelem cache je tedy urychlit přístup k datům z pomalého subsystému na rychlý subsystém uložením dat do této mezipaměti. Pokud se požadované informace nachází v cache, nemusí se čekat, než jsou data dopravena z pomalejšího média a požadovaná operace bývá vykonána mnohonásobně rychleji. Příkladem hardwarových komponentů, kde se cache využívá, je například procesor. Tato cache se nachází mezi procesorem a pamětí RAM, a její využití správnými programátorskými praktikami může extrémně ovlivnit rychlost, se kterou se budou tyto algoritmy vykonávat.

Mezi další HW komponenty obsahující cache patří například GPU, HDD a SSD. Cache ovšem nejsou omezené pouze na HW komponenty, jelikož se dnes používají i webové cache v prohlížečích, a také například DNS nebo P2P cache. Dalším speciálním druhem této komponenty je tzv. TLB, která bude dále více rozvedena.

3.1 CPU Cache

3.1.1 Historie

Vynalezení cache je jednou z nejdůležitějších událostí ve vývoje počítačů. Motivací pro její rozvoj byl fakt, že mikroprocesor začal být v polovině 80. let natolik rychlý, že jej začala paměť RAM zpomalovat. První možností, jak situaci vyřešit, bylo zrychlení celé RAM paměti. To sice technicky možné je, ale za cenu mnohonásobného zdražení. A tak se vyrazilo ekonomičtější cestou, kdy se k pomalé RAM paměti přidaly rychlé, malé paměti cache[5]. Později se cache umisťovala vedle procesoru, ale nakonec se s ním sloučila. Cache se postupně velice zdokonalovala a v dnešním světě počítačů si již život bez ní nelze představit. Obrázek 1 obecně zobrazuje, jaký byl rozdíl v růstu výkonu CPU a RAM v letech 1980 až 2010.

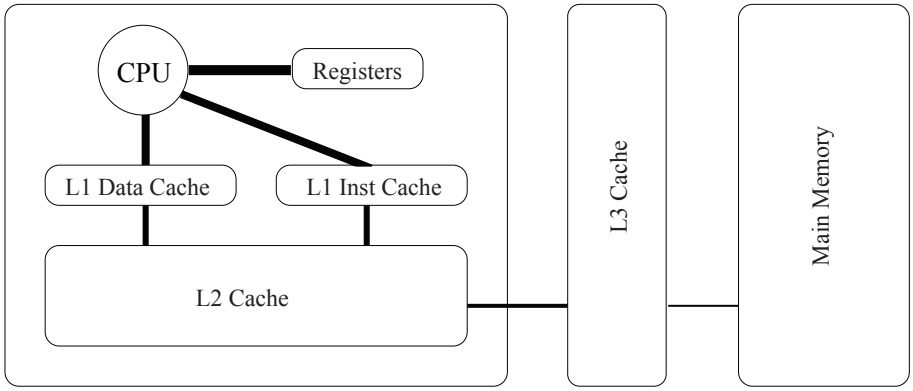


Obrázek 1: Vývoj CPU a RAM[6]

3.1.2 Fyzické uspořádání cache úrovní

Typický procesor obsahuje hlavní, *L1 cache*, která je umístěna přímo na procesoru. Tato cache se dále rozděluje na *L1 datovou cache* a *L1 instrukční cache*, které mají většinou stejnou velikost. Velikost této cache se pohybuje v řádu desítek kB pro každé jádro. Dále obsahuje *L2 cache*, která je pomalejší, ale zato má větší velikost. Tato paměť se také nachází přímo na čipu. Její velikost se různí podle toho, zda je přítomna i *L3 cache*. Pokud přítomna je, bývá její velikost typicky 256kB nebo 512kB pro každé jádro. Pokud není, může být i v řádech MB[7][8]. V běžných moderních procesorech se pak ještě nachází největší, L3 cache, která má velikost několik MB a je sdílená pro všechna jádra.

Nedílnou součástí CPU je pak sada *registrů*, neboli *Registers*, které slouží k uchovávání operandů a mezivýsledků. Obrázek číslo 2 popisuje paměťovou architekturu CPU, kde šířka čar značí přenosovou rychlost mezi jednotlivými komponentami.



Obrázek 2: Architektura cache[7]

Procesorová cache obsahuje kopie dat z hlavní paměti RAM, neboli *Main Memory*. Při transportu dat do cache se zde vytvoří záznam, tzv. *cache entry*. Tato data jsou uložena v takzvaných řádcích cache, neboli *cache lines*. Typicky se cache entry skládá ze tří základních součástí. Z adresy, odkud byla data zkopírována, z aktuálních dat, tedy cache line a z pomocných bitů. Tyto bity jsou například *valid* bit, který ukazuje, zda byla data korektně nahrána a *dirty* bit, který říká, zda byla data nahraná v cache změněna, a tyto změny se ještě neprojevily v původní paměti[9].

Celkový počet cache entries E_x lze vypočíst ze vztahu číslo 1, kde *x* je úroveň (level) cache, $|L_x|$ je celková velikost x-té cache a $|Line_x|$ je velikost jedné řádky L_x cache.

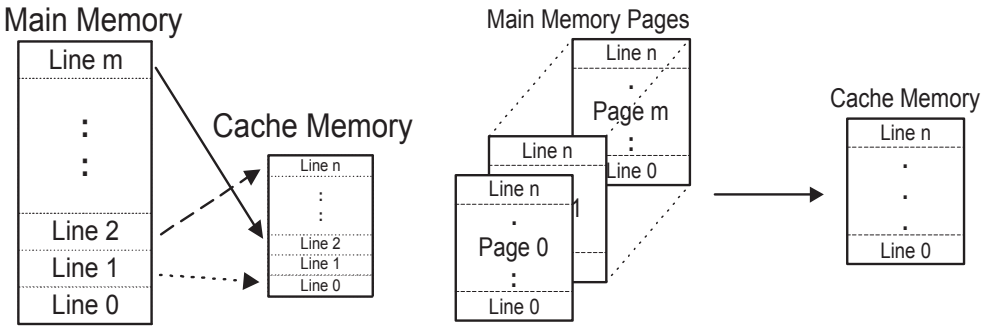
$$E_x = |L_x| \ / \ |Line_x| \tag{1}$$

3.1.3 Organizace načítání dat z RAM

Aby mohl procesor pracovat s obsahem paměti, musí si ji nejdříve přesunout do cache. Pokud načítáme data z paměti, je důležité definovat, jakými pravidly se bude řídit mapování dat z RAM do cache. V následujících řádcích budou popsány dvě metody, které se tímto zabývají.

3.1.3.1 Plně asociační přístup neboli *Fully - Associative*, je metoda, při které je cache i hlavní paměť rozdělena na řádky (cache lines) stejné velikosti. Jak napovídá název, lze jakoukoli řádku hlavní paměti mapovat na jakoukoli řádku paměti cache. Tato metoda se vyznačuje největším výkonem, protože lze každou adresu hlavní paměti zapsat kdekoli do cache. Problém ale způsobuje komplexita porovnávání všech řádek, a tak se tento typ hodí pouze na cache paměti malých velikostí[10].

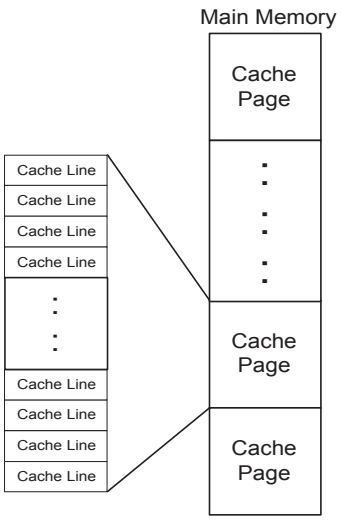
3.1.3.2 Přímé mapování neboli *Direct Mapping* je druhý způsob, který bude v této práci představen. Hlavní paměť je zde rozdělena na stránky, neboli *Main Memory Cache Pages*. Každá stránka má takovou velikost, jako je celková velikost cache. Rozdíl mezi přímým a plně asociačním přístupem je ten, že každá řádka paměti se může mapovat pouze na jednu konkrétní řádku cache. Například řádka 1 stránky 1 se může mapovat pouze na řádku 1 v cache. Tímto odpadá režie nahrazování stránek a snižuje se náročnost na implementaci a cenu. Nevýhodou však je menší flexibilita a výrazně nižší výkonnost paměti, a to hlavně v případě, kdy je potřeba přeskakovat mezi jednotlivými stránkami. Obrázek číslo 3 pak obě metody popisuje a obrázek číslo 4 pak vyobrazuje rozdělení paměti na stránky.



(a) Plně asociační přístup

(b) Přímé mapování

Obrázek 3: Načítání paměti z cache[10]



Obrázek 4: Rozdělení paměti(Main Memory) na stránky[10]

3.1.4 Metody pro nahrazování dat v cache(*replacement policy*)

Aby se vytvořilo místo pro nový záznam v cache, je třeba smazat některý ze stávajících záznamů. Při přímém mapování volba smazaného záznamu není problém, protože je vždy jasné, kam se nový záznam zapíše. Nicméně při plně asociačním mapování je správný algoritmus, řešící tzv. replacement policy faktor, který může výrazně ovlivnit výkon. V ideálním případě by se vždy uvolňoval takový záznam, který se již v budoucnu nebude používat. Předpovídat budoucnost ale stále není možné, a tak optimálního algoritmu není možno dosáhnout.

Existují ovšem metody, které se o to více či méně úspěšně pokoušejí. Mezi tyto algoritmy patří například metoda náhodného výběru, *First In-First Out* , neboli *FIFO* nebo metoda *Least Recently Used*, neboli *LRU*. U náhodného výběru (*random selection*) při přepisování řádky hraje roli náhoda, zatímco metoda FIFO funguje na principu fronty. U metody LRU je vybrán ten záznam, který je v cache a byl nejdéle nevyužit. Tato metoda se také osvědčila jako nejvíce efektivní[11].

3.1.5 Cache hit a Cache miss

Cache hit a *cache miss* jsou další termíny, které je nutno definovat. Jestliže procesor potřebuje data, nejdříve se podívá do cache, a pokud zde byl záznam nalezen, došlo ke cache hit. Samozřejmě není možné, aby byla veškerá data z hlavní paměti uložena v cache, a

proto dochází ke cache misses, jejichž počet lze do jisté míry eliminovat. [1]

Jestliže se data nacházejí v L1 cache, mluvíme o L1 cache hit a procesor může pokračovat v práci v plné rychlosti. V případě, že se data nenacházejí v L1 cache, nastal L1 cache miss, a procesor zkontroluje další cache level, tedy L2 cache. Pokud se data nachází v L2 cache, pak hovoříme o L2 cache hit. V případě, že se data v cache nenachází vůbec, je nutno je načíst z hlavní paměti RAM. Počet L1 cache misses lze tedy definovat vztahem číslo 2.

$$|L1_{misses}| = |L2_{hits}| + .. + |LLC_{hits}| + |LLC_{misses}| \tag{2}$$

Kde *LLC* značí *Last-Level Cache*, tedy cache poslední úrovně. V případě, že tedy dojde ke cache miss, dochází ke zpoždění, které je rovno *cache latency*. Latency lze definovat jako čas, buď to v CPU cyklech nebo nanosekundách, který CPU ztratí z důvodu jednoho cache miss. [2] [3]

3.1.5.1 Rozdělení cache misses Nespočet výzkumů týkajících se cache bylo provedeno pro nalezení ideálního poměru mezi velikostí, asociativitou a počtem a velikosti bloků, aby se co nejvíce snížil počet misses. Jedním z nejvýznamněnších badatelů v této oblasti je *Mark Hill* [13], který rozdělil procesorové cache misses do následujících kategorií[14][15].

- Povinné(*compulsory*) misses: Jedná se o misses, které jsou způsobeny dotazem na místo v paměti, které ještě předtím nebylo požadováno. Těmto misses nelze nikdy úplně předejít, nicméně se dají omezit pomocí metody zvané *prefetching*, která bude zmíněna v další kapitole, nebo pomocí větší řádky cache, což se dá také považovat za jistý druh prefetche.
- Kapacitní(*capacity*) misses: Jsou založeny na faktu, že cache má konečnou velikost a nelze tedy celou paměť mapovat do cache.
- Konfliktní(*conflict*) misses: Misses, ke kterým došlo tím, že byla v budoucnu potřebná řádka v cache nahrazená řádkou novou. Tento druh se dá dále rozdělit do dvou podkategorií.
  - Mapovací(*mapping*) misses, kterým se nelze vyhnout v rámci používané asociativity(viz kapitola přímé mapování)
  - Nahrazovací(*replacement*) misses, které vznikají špatnou volbou nahrazení nahrazovací metody, a lze je tedy pouze omezit použitím co nejvhodnějšího nahrazovacího algoritmu.

[1] Některé paměťové regiony cachovat nelze, což je otázka implementace OS a programátor toto nemůže ovlivnit[12].

[2] Všechna zpoždění, jak cache tak i hlavní paměti, se liší podle modelu procesoru, a samozřejmě čím menší, tím lepší.

[3] Procentuální vyjádření úspěšnosti, kdy jsou data nalezena v cache, tedy cache hit, vyjadřuje tzv. *cache hit rate* a bývá okolo 95 procent.

Dále lze cache misses rozdělit podle toho, zda vznikají během čtení, nebo zápisu do cache.

- Cache miss během zápisu (*write miss*) způsobuje nejkratší odezvu, protože zápis se může uložit do fronty, a procesor může pokračovat v práci, dokud není fronta zaplněna.
- Cache misses vzniklé během čtení (*read miss*) lze dále rozdělit do dvou subkategorií, a to na read miss z instrukční cache a read miss z datové cache. Cache miss read z instrukční cache způsobuje nejdelší zpoždění, protože procesor nebo alespoň vlákno musí čekat, dokud není instrukce dodána z paměti. Oproti tomu cache miss read z datové cache způsobuje menší zpoždění, protože instrukce nezávislé na datech mohou pokračovat, a závislé instrukce budou dokončeny později.

3.1.6 Neprogramátorské metody řešící cache misses

Jednou z těchto metod je například prefetching. Jednoduše lze říci, že nastává v případě, kdy procesor načte v budoucnu požadovaná data z hlavní paměti spolu s dalšími, která jsou aktuálně nepotřebná. Pokud poté bude procesor tato data potřebovat, budou v ideálním případě nalezena v cache a nedojde ke zdržení procesoru[16]. Další variantou je minimalizovat dopad cache miss až poté, co k němu došlo. Jednou z těchto technologií je například *Hyper-Threading* od společnosti Intel, a funguje na základě rozdělování zdrojů mezi dvě vlákna každého jádra. Pokud tedy jedno jádro čeká z důvodu cache misses na data z paměti, druhé jádro si vypůjčí výpočetní výkon druhého[17].

3.1.7 TLB

Procesorové procesy pracují s daty na virtuálních adresách. Pokud proces potřebuje použít data v paměti, kde jsou uchovávána na fyzických adresách, je nutno virtuální adresu převést na fyzickou. V takovém případě je úkolem operačního systému, aby procesu tuto adresu zajistil, tedy adresu, kde jsou uložena skutečná data. Tuto funkci urychluje TLB, neboli *Translation Lookaside Buffer*, což je další druh procesorové cache a konkrétně je součástí MMU, neboli *Memory Management Unit*, která tento překlad vykonává.

TLB tedy obsahuje část naposledy použitých adres načtených z tzv. *page table*, což je datová struktura, která mapuje všechny virtuální adresy na fyzické. Tato cache je plně asociační a stejně jako procesorovou cache ji lze rozdělit na omezený počet záznamů, neboli entries. Každý záznam obsahuje VPN, neboli *Virtual-Page-Number* obsahující virtuální adresu, PFN, neboli *Page-Frame-Number*, ze které lze zjistit hledaná fyzická adresa a další bity. V následujícím odstavci je zjednodušeně popsán algoritmus hledání v TLB.

Nejdříve se zjistí, zda-li TLB obsahuje záznam s aktuálně překládanou virtuální adresou. Pokud ano, nastal TLB hit a celá operace je velice rychlá. V opačném případě dochází k TLB miss a je třeba vykonat určitou práci navíc. Nejdříve se prochází page table a po nalezení se hledaný záznam přenese do TLB. Tento proces se pak zopakuje s již aktualizovanou TLB tabulkou[18].

4 Tvorba optimalizovaných algoritmů

4.1 Motivace pro tvorbu optimalizovaných algoritmů

Na následujícím zjednodušeném příkladu se pokusíme demonstrovat, proč je procesorová cache důležitá. Představme si, že procesor musí nahrát data z L1 cache stokrát za sebou. Například na jednom z testovacích procesorů trvá přístup k datům v L1 cache 1ns. Procesor tedy potřebuje 100ns, aby tuto operaci vykonal. Moderní procesory mají L1 cache hit rate okolo 95%. Zjednodušeně můžeme tedy říci, že prvních 95 přístupů se vykonalo za 95ns, ale posledních 5 záznamů nebylo nalezeno v L1 cache. Nastal tedy L1 cache miss. Zbylá data byla nalezena v L2 cache, což vyústilo v L2 cache hit. L1 cache miss zpoždění je na mém stroji 6ns. Z tohoto lze odvodit, že přístup k posledním 5 záznamům trval procesoru 30ns, a to za předpokladu, že data byla nalezena v L2 cache.

Pokud by data nebyla nalezena ani v L2 cache, musel by procesor sáhnout do paměti RAM, ze které trvá dopravit data do procesoru přibližně 30ns. Z tohoto faktu lze vyvodit, že načtení zbylých pěti záznamů z paměti by procesoru trvalo déle, než načtení předchozích 95[19].

4.2 Nástroje pro optimalizaci algoritmů vůči cache a TLB

Pokud chceme optimalizovat algoritmy s přihlédnutím ke cache, je nutné si o ní nejdříve zjistit základní parametry. Velikost většinou není problém ověřit například na internetových stránkách výrobce konkrétního procesoru, ale většinou lze, stejně jako v této práci, využít znalosti podrobnějších parametrů.

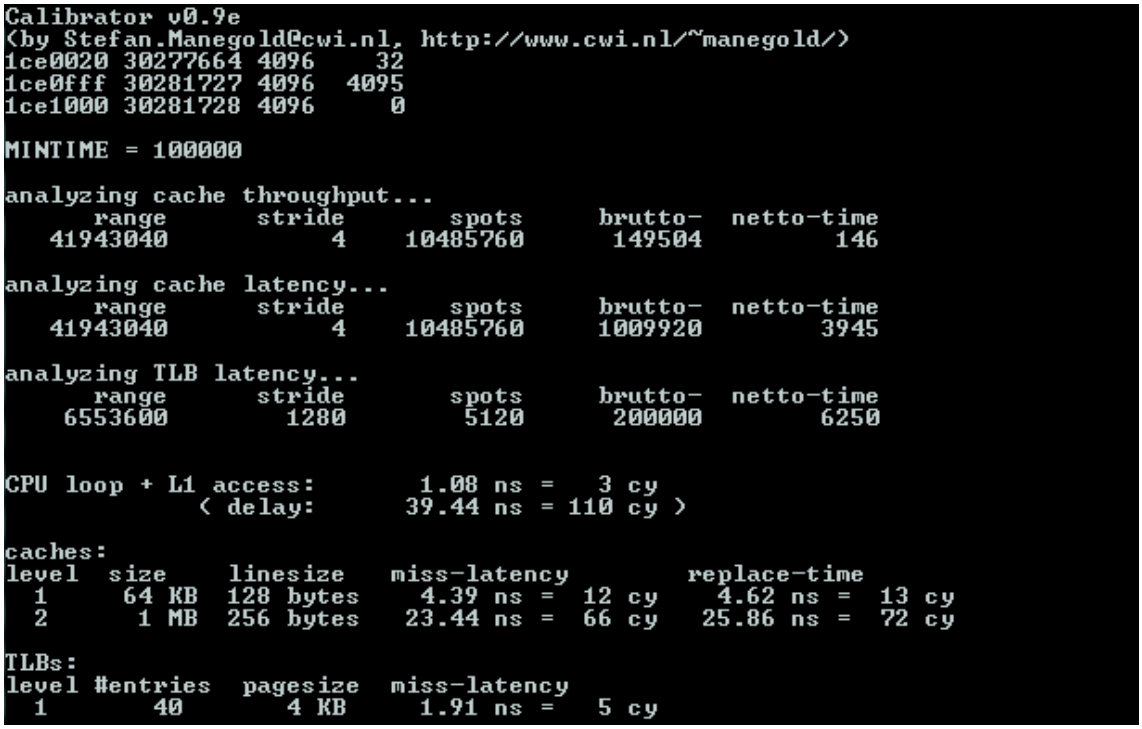
Pro tyto účely byl použit nástroj *Calibrator*. Jedná se o malý program napsaný v jazyce C, který zanalyzuje procesor a je schopen zjistit následující parametry[20]:

- Počet cache úrovní a jejich velikosti
- Velikost řádky a počet záznamů
- Miss latency
- Počet TLB úrovní, jejich velikost a latency

Na obrázku5 vidíme ukázkový výstup programu Calibrator.

Pro ověření výsledků a správnosti algoritmů bylo mimo měření doby běhu programu také měřen počet cache misses. Pro tento účel posloužil program *AQTime* od společnosti *SmartBear Software*. Tento nástroj umí měřit cache misses pro druhý level cache, tedy L2[21].

Poznámka 4.1 V 64-bit operačních systémech windows lze měřit pouze skutečnou dobu běhu programu, jelikož 64-bit systémy od verze Windows XP mají tzv. *Kernel Patch Protection*, která brání záplatování kernelu. Pro zpřístupnění všech profilovacích nástojů včetně měření cache misses bylo třeba aktivovat debug mód v operačním systému[22].



Obrázek 5: Ukázkový výstup nástroje Calibrator

Pro Windows 7 lze tento mód aktivovat spuštěním následujících příkazů:

Bcdedit /debug ON
Bcdedit /dbgsettings SERIAL DEBUGPORT:1 BAUDRATE:115200 /start AUTOENABLE

Pro deaktivaci pak stačí zadat příkaz:

Bcdedit /debug OFF

V obou případech je nutno pro projevení změn restartovat počítač.

4.3 Programátorské techniky cache optimalizací

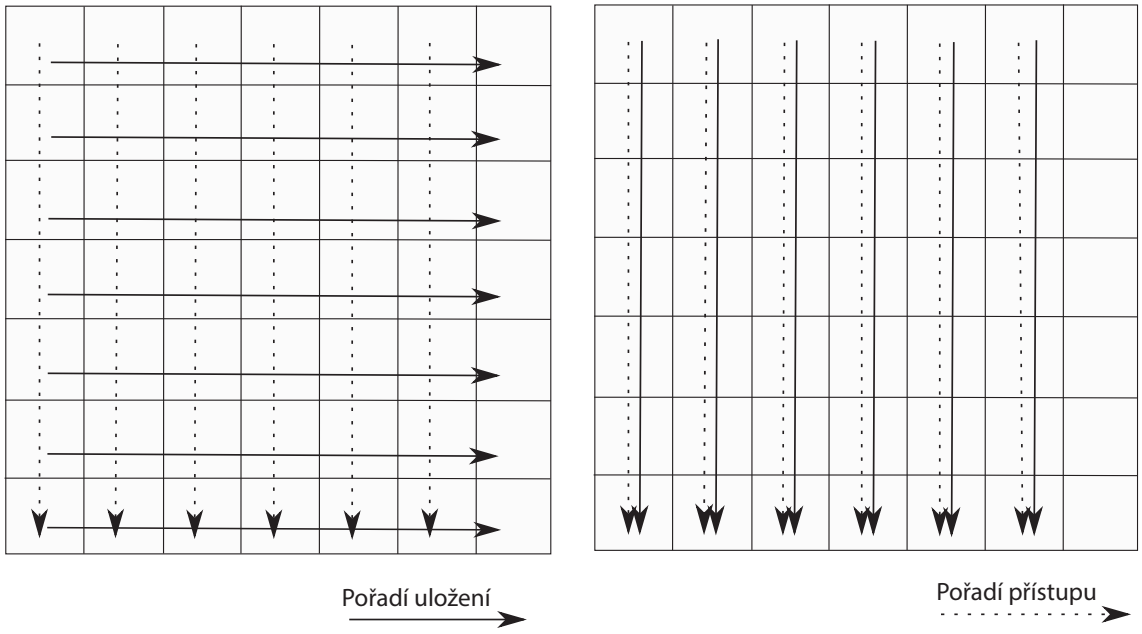
Jedná se o optimalizace, které může programátor využít k urychlení provádění kódu. Tyto techniky lze rozdělit do dvou kategorií, a to na optimalizace *přístupu k datům* a optimalizace *rozložení dat*.

- První kategorií je změna přístupu k datům. Tyto optimalizace většinou fungují na principu změny pořadí, ve kterém jsou iterace zanořených cyklů prováděny. Efekt těchto změn se nazývá vylepšená *dočasná lokalita dat*. Mezi tyto optimalizace patří například *prohození cyklů* nebo *splynutí cyklů*.

– Prohození cyklů je transformace, která prohodí dva přilehlé cykly. Obecně se dá říci, že ji lze použít tam, kde nezáleží na pořadí, ve kterém jsou cykly vykonávány. Příkladem takového algoritmu může být například sčítání matic, kde je tento problém demonstrován na výpisu kódu číslo 4, kde podtržené části řádků značí rozdíly mezi implementacemi, a na obrázku číslo 6.

Algoritmus 4: Ukázka prohození cyklů

```
1  for i = 0 to N do                                   // Před prohozením
2  |   for j = 0 to N do
3  |   |   C[i][j] = A[i][j] + B[i][j];
4  |   end
5  end
6  for j = 0 to N do                                   // Po prohození
7  |   for i = 0 to N do
8  |   |   C[i][j] = A[i][j] + B[i][j];
9  |   end
10 end
```



(a) Před transformací (b) Po transformaci

Obrázek 6: Ilustrace prohození cyklů

  - Splynutí cyklů je technika, která spojí dva po sobě jdoucí cykly mající stejný počet iterací do jednoho cyklu. Její použití je možné za předpokladu, že výsledky druhého cyklu nejsou závislé na výsledky prvního. Kromě jasné úspory času způsobené odstraněním jednoho či více cyklů dojde také k lepší dočasné lokalitě dat.

- Druhou kategorií je změna rozložení dat. Tyto optimalizace mění způsoby, jakými jsou data uspořádána v paměti. Zaměřují se zejména na to, aby omezili konfliktní cache misses a zlepšili prostorové uspořádání kódu.

  - *Array padding* mezi dvě na-alokovaná pole vloží pole další, které se říká vycpávka, neboli *pad*. Velikost této vycpávky by měla ideálně být o velikosti jedné cache řádky.
  - *Splynutí polí* vylepšuje prostorovou lokalitu mezi prvky více polí tím, že spojí více jednorozměrných polí do jednoho vícerozměrného. Hodí se zejména v případě, kdy by byly pole na-alokovány v paměti daleko od sebe, ale přistupovalo by se k nim společně.

  Dalším typem optimalizace je přihrádkování, které bude zmíněno v následující kapitole a které bylo použito při implementaci algoritmů.

5 Implementované algoritmy

5.1 GRACE hash-join

Prvním z algoritmů, které byly v rámci této práce implementovány je *GRACE hash-join*, neboli *partitioned hash-join*. Jedná se o optimalizovanou variantu hash-joinu, která využívá jedné z cache optimalizací, a to konkrétně přihrádkování, což je technika, kdy je datová struktura, jejíž celková velikost se nevejde do cache, rozdělena do struktur menších. Tento algoritmus je také popsán na výpisu číslo 5.

Stejně jako hash-join, který byl zmíněn kapitole 2, i partitioned hash-join se skládá ze dvou fází, a to z build fáze (viz řádky číslo 1 a 8) a probe fáze (viz řádek 15). U GRACE hash-joinu je součástí build fáze navíc přihrádkování větší relace. Řádky, které jsou odlišné od obyčejného hash-joinu jsou zde podtrženy.

Algoritmus 5: GRACE hash-join

```
   input  : relace R, relace S
   output: relace Q
 1 for each r v R do                                  // Build fáze relace R
 2 |   vypočti hash pole pro r;
 3 |   vlož r do vypočteného hash pole;
 4 |   if hash pole je plné then
 5 |   |   zvětši hash pole;
 6 |   end
 7 end
 8 for each s in S do                                 // Build fáze relace S
 9 |   vypočti přihrádku pro s;
10 |   vlož s do vypočtené přihrádky;
11 |   if přihrádka je plná then
12 |   |   zvětši přihrádku;
13 |   end
14 end
15 for each přihrádku relace S do                          // Probe fáze
16 |   for each záznam s v přihrádce do
17 |   |   vypočti hash pole pro s;
18 |   |   for each r ve vypočteném hash poli do
19 |   |   |   if r == s then
20 |   |   |   |   vlož r a s do Q;
21 |   |   |   end
22 |   |   end
23 |   end
24 end
```

Přihrádkování druhé relace (relace B) samozřejmě znamená mnoho operací navíc, protože podle velikosti relací, je třeba rozdělit, a tím pádem sekvenčně projít relace i velmi velké velikostech. Zvýšené časové náklady na build fázi ale nakonec eliminuje jeho výhoda, která se projeví až v probe fázi.

Tato výhoda zpočívá právě v omezení cache misses, které je způsobena tím, že můžeme mít najednou celé přistupované části hash tabulky relace A v cache, a také jsme tím částečně uspořádali větší relaci podle spojovacího atributu. Podmínkou pro toto seřazení je přihrádkování relace podle spojovacího atributu, a ne podle primárního klíče.

Nevýhodou těchto optimalizací je samotný proces přihrádkování, který zvyšuje nejen časovou náročnost, ale také výrazně zvyšuje využití paměťových zdrojů, a také nutnost pomocné datové struktury, která by si uchovávala aktuální počet prvků a maximální počet prvků v každé přihrádce, respektive hashovaném poli tabulky.

Během implementace bylo nejdříve pro tyto účely použito dvourozměrné pole, ale následnou transformací na pole jednorozměrné jsme získali souvislý blok dat, který se načte do řádky cache první referencí na toto pole, čímž v budoucnu omezíme počet cache misses. Konkrétní výsledky této optimalizace budou prezentovány v sekci 7.3.

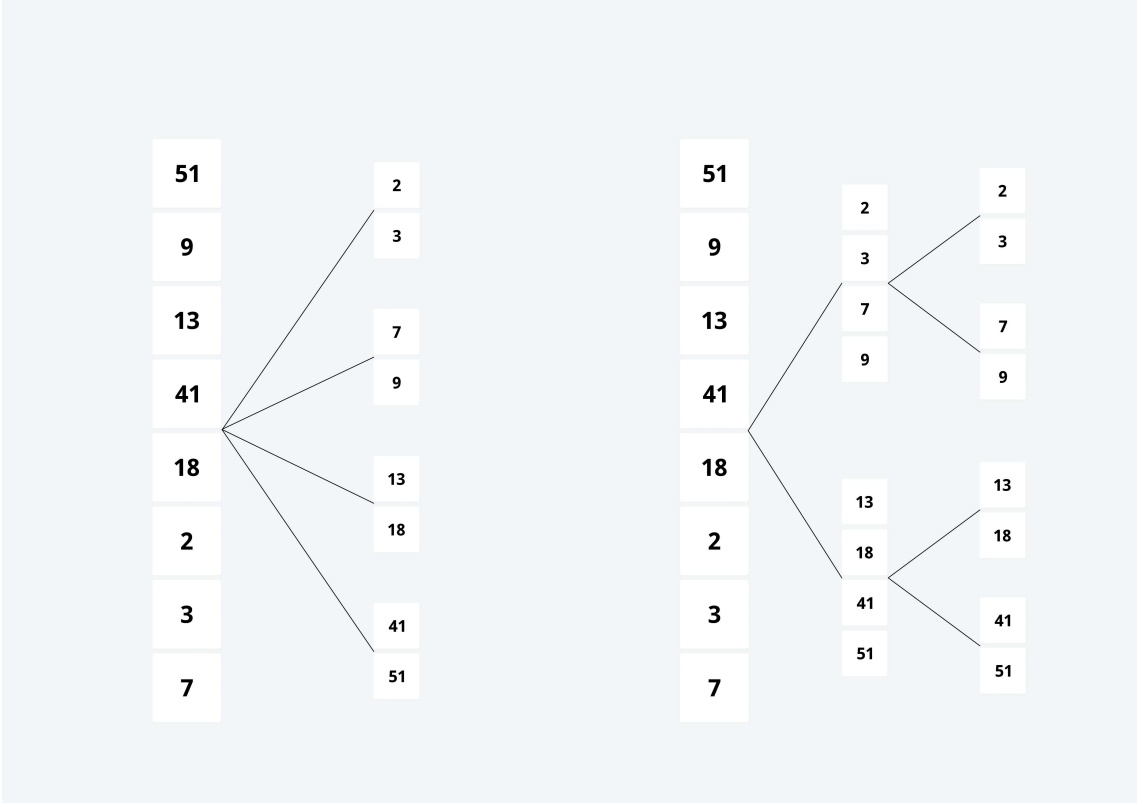
5.2 Radix-Cluster

Druhý představovaný algoritmus se nazývá radix-cluster hash-join. Z názvu lze vyčíst, že se jedná o další vylepšení algoritmu hash-join, přesněji již optimalizovaného GRACE hash-joinu. Oproti GRACE hash-joinu je proces tvorby hashovací tabulky vykonáván ve více etapách, jejichž počet lze označit písmenem P. [4] Když tvorba hashovací tabulky začíná, je celá relace považována za jednu přihrádku a je prvním průchodem rozdělena do jednotlivých polí tabulky. V dalším průchodu se pak vezme každé nové pole, které se následně opět rozdělí do polí nových. Více průchodové rozdělování se aplikuje pouze na menší relaci, která má po build fázi mnohonásobně větší počet přihrádek, než relace větší. Abychom tedy dosáhli u větší relace stejného efektu, jako u relace menší, musela by relace mít enormní velikost, řádově stovky miliónů až miliardy.

Pro názornost ukazuje rozdíl mezi GRACE hash-joinem a radix-cluster hash-joinem obrázek číslo 7.

Důležitou vlastností, a smysl celého algoritmu je ten, že najednou hashujeme pouze do omezenému počtu přihrádek, čímž se omezí množství přistupovaných regionů v paměti. Přesněji, pokud nový počet přihrádek z jediné přihrádky bude menší, než počet dostupných řádek cache, výrazně omezíme cache misses build fáze, a navíc, pokud tento počet bude menší, než počet TLB entries, eliminujeme tím i počet TLB misses.

[4] Pokud je P = 1, vznikne z algoritmu jednoduchý GRACE hash-join.



(a) Jedno-průchodový GRACE hash-join (b) Dvou-průchodový radix-cluster

Obrázek 7: Srovnání GRACE hash-joinu s radix-clusterem

6 Popis implementace a vyčíslení potřebných vstupů

Pro implementaci byl z výkonnostních důvodů zvolen jazyk *C++* a programováno bylo v prostředí *Microsoft Visual Studio 2012 Professional*.

Pro přípravu dat byly použity mimo základních knihoven také externí knihovny *Boost*, které tato práce využívá pro generování náhodných čísel o velikosti v řádech miliónů, jelikož základní funkce $rand()$ z knihovny $stdlib.h$ generuje pouze čísla do velikosti *RAND_MAX*, jehož hodnota je garantována pouze na velikost 32 767[24][25].

Všechny algoritmy byly také zpracovávány s důrazem na genericitu, a díky použití šablon můžeme algoritmus použít pro jakýkoliv datový typ. Jediná podmínka zde spočívá v tom, že je nutné si nadefinovat svou vlastní hashovací funkci, která je, samozřejmě, jiná při práci s typem int, který byl ve všech testech používán, a u kterého lze použít operaci dělení, modulo, popřípadě bitové operace. Tato volba hashovací funkce se realizuje pomocí ukazatele na funkci, který je podle potřeby předáván algoritmu.

Předávání hashovací funkce bylo ještě dále optimalizováno na úkor jednoduché rozšiřovatelnosti. I přesto, že byla funkce implementována jako inline, docházelo z důvodu velkého množství volání předávané funkce, které je v řádu miliónů, k časovému zpomalení. Tento problém byl vyřešen předáváním celé funkce pro build fázi, respektive probe fázi a tím se předávaná funkce volá pouze jednou.

6.1 Vyčíslení potřebných vstupů

Pro úspěšné vykonání algoritmu je potřeba stanovit následující parametry:

- Předpokládaný počet prvků jedné přihrádky obou relací
- Počty přihrádek obou relací

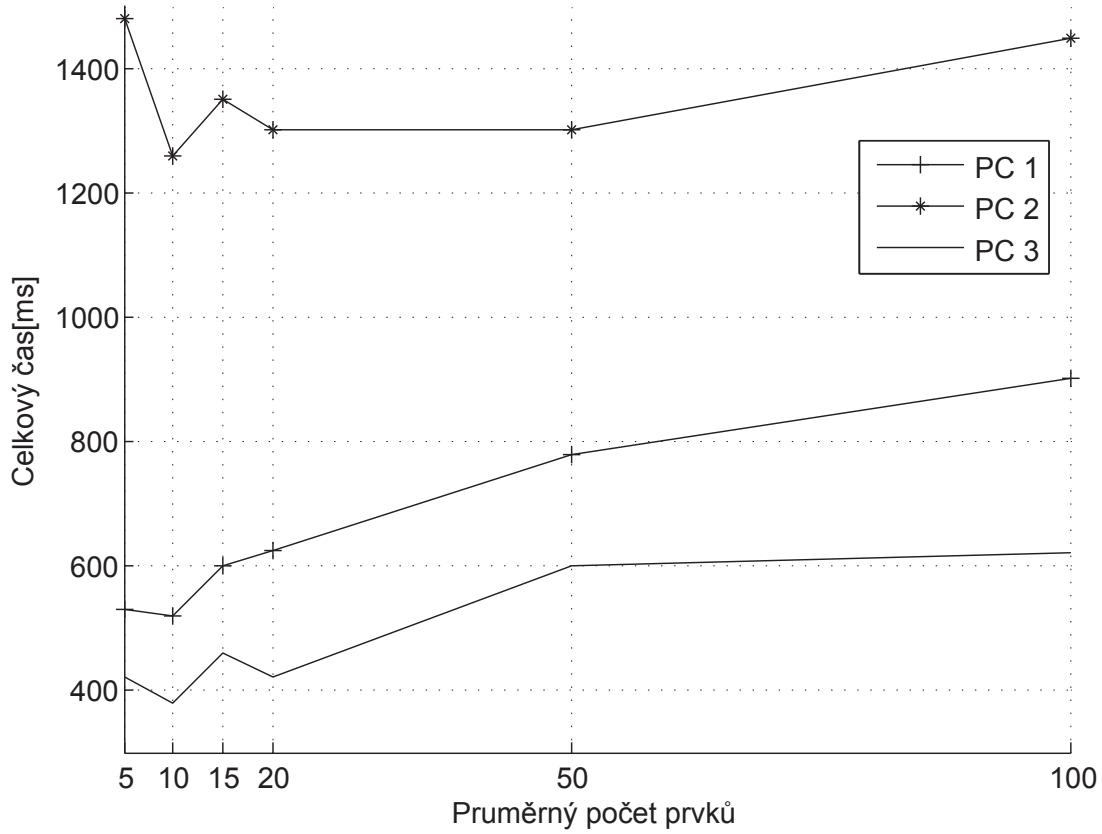
V případě, že se jedná o radix-cluster algoritmus, je zapotřebí také stanovit další hodnoty:

- Počet nových přihrádek vzniklých každým průchodem z jednotlivé přihrádky
- Počet průchodů build fáze

6.1.1 Předpokládaný počet prvků v přihrádkách

Určení předpokládaného počtu prvků se liší v tom, zda se jedná o menší relaci, nebo o relaci větší. V prvním případě jsme tento problém vyřešili testováním na všech počítačích. Tímto testováním jsme se dostali na ideální hodnoty deseti až třinácti prvků na jednu přihrádku, a tak byla hodnota 10 použita jako referenční hodnota pro všechny algoritmy. Testování probíhalo měřením celkového algoritmického času s tím, že všechny vstupní hodnoty byly zafixovány a měnili jsme pouze hledaný počet prvků. Graf č. 1 pak toto měření v závislosti na čase zobrazuje.

Poznámka 6.1 Test probíhal na algoritmu hash-join, který byl dále modifikován tak, aby bylo možno přesně specifikovat velikost jednoho hashovaného pole.



Graf 1: Test počtu prvků v hashovaném poli

Tento počet byl dále upraven, aby byl počet hashovaných polí násobkem čísla dva, a tím se mohlo využít méně náročných bitových operací.

Vyřešit tuto hodnotu u větší relace je už o něco jednodušší. Jedná se zejména o to, aby se celá přihrádka vešla do cache paměti, pro kterou chceme optimalizovat. Celkový počet prvků lze tedy získat vztahem číslo 3, kde $|cache\_size|$ je velikost cache, pro kterou chceme optimalizovat(v bytech) a *sizeof(T)* je počet bytů, který daný datový typ zabírá v paměti.

$$|Bbucket| = |cache\_size|/sizeof(T) \tag{3}$$

V ideálním případě by tyto hodnoty měly samozřejmě být takové, aby každá přihrádka byla po buid fázi úplně zaplněna, a zároveň nedocházelo k volání funkce *enlarge()*, která nám navýší naplněné hashované pole, respektive přihrádku větší relace. Tohoto se ovšem dá bez předchozích průchodů, při kterých bychom byli schopni zjistit, jakým způsobem jsou prvky v závislosti na spojovací atribut rozmístěny, docílit jen těžko.

Byla tedy použita silně optimistická metoda, při které se počítá s tím, že jsou prvky rozmístěny rovnoměrně. I přesto však algoritmy používají dva způsoby, jak řešit problémy týkající se nerovnoměrného rozmístění. Zaprvé, jestliže dojde k naplnění přihrádky, je zavolána výše zmíněná metoda *enlarge()*. Tato konstrukce umožňuje algoritmu aby pokračoval dále, nicméně dochází k rapidnímu snížení výkonu a také k plýtvání s operační pamětí. Aby již při malé odchylce od ideálního stavu nedocházelo k tomuto problému, do každé přihrádky je navíc přidán určitý počet prvků navíc.

Tento počet lze vyjádřit vztahem 4, který tento problém vyjadřuje pro relaci B. [5]

$$Bbucket_{added\_size} = |Bbucket| * (scattery\_coeficient/100) + 3 \quad (4)$$

Proměnná *scattery_coeficient* představuje vstupní hodnotu algoritmu, kterou může uživatel algoritmu sdělit, jak moc nerovnoměrně jsou data rozmístěna. Lze tedy říci, že tento koeficient procentuálně navyšuje velikost každé přihrádky. Zajímavá vlastnost této optimalizace je ta, že tím nijak neutrpí výkon probe fáze, protože v případě, že nebude navýšená velikost dosažena, je stále procházena pouze naplněná část, což zaručuje znalost aktuálního počtu prvků v přihrádce v pomocné struktuře.

Hodnota 3 se může pak zdát bezvýznamná, opak je ale pravdou. Tato hodnota slouží k tomu, abychom byli schopni navýšit takové přihrádky, jejichž velikost je příliš malá a navýšení o několik procent by nemělo žádný význam.

Jedinou nevýhodou, kterou tato operace přináší, je případné plýtvání paměti. Je potom na uživateli, který podle svého uvážení musí rozhodnout, jestli navýšení velikosti o desítky či stovky procent může pro jeho paměťové zdroje způsobit markantní zatížení.

6.2 Počet přihrádek

Když nyní známe hodnoty vypočtené v přechozích odstavcích, vypočíst počty bucketů již není problém. U menší relace určíme takový násobek čísla dva, aby byl celkový počet záznamů děleno tímto počtem co nejblíže číslu 10, a u relace větší vydělíme celkový počet prvků relace počtem prvků jedné přihrádky.

6.3 Vyčíslení parametrů pro radix-cluster

V případě radix-cluster joinu nám nyní zbývá zjistit počet nových hashovaných polí, které vzniknou každým průchodem z každého původního pole a počet průchodů. První zmiňovaná hodnota je jeden ze vstupních parametrů a musí se jednat o násobek čísla dva. Počet průchodů je pak nejnižší mocnina předchozí hodnoty, která je větší nebo rovna celkovému počtu hashovaných polí. Jednotlivé výsledky pro různé volby těchto hodnot lze vyčíst z grafů v kapitole 7.

[5] Pro relaci A platí stejný vztah s tím rozdílem, že místo velikosti jedné přihrádky relace B použijeme velikost jednoho hash pole relace A.

7 Testování algoritmů

V této kapitole budou probírány testy pro jednotlivé algoritmy. Nejdříve se podíváme na to, jaké výhody přineslo převedení dvourozměrné pomocné struktury na strukturu jednorozměrnou. Dále budeme testovat průběh a chování GRACE hash-join algoritmu tím, že ponecháme stejné vstupní hodnoty, ale budeme měnit počet přihrádek relace větší. Z měření pak budou vyvozeny zajímavé důsledky.

V dalším testu pak porovnáme výkon GRACE hash-joinu s obyčejným hash-joinem. Tento test bude prováděn s algoritmem optimalizujícím L1 cache, jelikož předchozí test ukázal, že je tato volba efektivnější..

Cílem posledních testů pak bylo zjistit, jak lze GRACE hash-join algoritmus zefektivnit využitím více průchodového radix-cluster hash-joinu.

Ve všech testech se také měřil pouze celkový čas build a probe fáze. Přípravné výpočty, alokace a následné dealokace součástí tohoto měření nebyly.

7.1 Popis testovaných hodnot

V experimentech byla použita vlastní testovací data, jejichž struktura zde bude nyní popsána.

Obě vstupní relace byly reprezentovány jednoduchým jednorozměrným polem objektů. Tato pole byla během testování relace A a relace B s kardinalitou N:1. Jeden prvek relace A tedy může mít 0 až N odpovídajících prvků z relace B a každý prvek relace B měl vždy odpovídající prvek v relaci A. Všechny testy probíhaly s náhodně rozmístěnými daty a s malým rozptýlením mezi jednotlivé cizí klíče. Dále byla data taková, že se každý hráč mapoval na jeden ze serverů.

Pro zjednodušení bylo pracováno s předpokladem, že spojovací atribut je primární klíč relace jedné, a cizí klíč relace druhé. Díky tomu nám stačí pracovat pouze s poli spojovacích atributů, a s polem primárního klíče relace druhé, který potřebujeme pro jednoznačnou identifikaci záznamu. Dále také předpokládejme, že menší relace je ta relace, ze které známe pouze primární klíč a že všechny atributy, se kterými se pracovalo byly typu int.

Poznámka 7.1 Relace *A* a relace *B* byla každá reprezentována jako třída, kde před každým spuštěním algoritmu docházelo k extrakci potřebných atributů. Tato extrakce nebyla součástí našich měření.

7.2 Seznam testovacích strojů

Pro ověření korektnosti algoritmů bylo prováděno testování na třech různých počítačích, jejichž využití je u každého odlišné.

Prvním počítačem, na kterém byly prováděny testy je ultrabook Acer Aspire S3, označený v tabulce 1 jako PC1. Tento počítač obsahuje dvoujádrový procedor Intel s L1, L2 i L3 cache. L1 má velikost 128KB pro instrukční i datovou cache, L2 cache pak 512KB.

	PC1	PC2	PC3
CPU	$IntelCorei3-3217U$	AMD Athlon II X2 240	$IntelXeonCPUX5670$
CPU_{speed}	$1.80GHz$	$2.80GHZ$	$2.97GHZ$
RAM	$4.00GB$	$4.00GB$	$92.00GB$
RAM_{lat}	$29ns$	$27ns$	-
$L1_{size}$	$128KB$	$64KB$	32KB
$L1_{lat}$	$1ns$	$4ns$	-
$L1_{entries}$	500	500	-
$L2_{size}$	$512KB$	$1MB$	256KB
$L2_{lat}$	$5ns$	$23ns$	-
$L2_{entries}$	1000	1000	-
$L3_{size}$	$3MB$	–	12MB
$L3_{lat}$	$23ns$	–	-
$L3_{entries}$	12000	–	-
$TLB_{pagesize}$	$4KB$	$4KB$	-
$TLB_{entries}$	64	40	-
TLB_{lat}	$8ns$	$2ns$	-

Tabulka 1: Seznam testovaných procesorů

L1 i L2 má každé jádrou své vlastní, L3 cache o velikosti 3MB je pak sdílená. CPU dále také disponuje technologií Hyper-Threading.

Dalším z počítačů je běžné stolní zařízení staršího data výroby, v tabulce 1 označený jako PC2. Jeho AMD procesor se začal vyrábět v polovině roku 2009 a v době, kdy tato práce vznikla byl již šest let starý. Procesor je dvoujádrový a každé jádro má svou vlastní cache, L1 a L2. L3 cache zde chybí, protože jak bylo řečeno v předchozích kapitolách, starší modely běžných osobních počítačů ji neobsahují. Tím pádem obsahuje ve srovnání s ostatními procesory velkou L2 cache, jejíž velikost je 1MB. L1 cache pak nabízí pro každé jádro 64KB datovou a instrukční cache.

Posledním počítačem je pak šestijádrový procesor Intel Xeon, označený v tabulce 1 jako PC3. K tomuto počítači byl v rámci tématu umožněn přístup vedoucím bakalářské práce. Všechna jádra mají k dispozici 12MB sdílené L3 cache, každé jádro pak 256KB L2 cache a také 32KB instrukční a datové L1 cache. Hyper-Threading je samozřejmostí.

Více informací o všech procesorech pak obsahuje tabulka číslo 1. U třetího procesoru nebylo možné přesně určit některé parametry a také nebylo možno měřit cache misses, jelikož tento model CPU není v programu AQTime podporován[26]. Stejně tak nebylo možno měřit cache misses na PC číslo 1, jelikož zde také AQTime nepracoval korektně.

Některé další údaje, jako jsou například délky zpoždění je pak nutné brát jako orientační.

7.3 Test transformace pomocné struktury

Tento test byl proveden za použití GRACE hash-joinu na PC2 a popisuje situaci, která byla zmíněna v sekci 5. Vstupní relace měly velikost jeden milión prvků pro relaci A a pět miliónů prvků pro relaci B. Tato na první pohled nedůležitá optimalizace snížila pro testovanou sestavu časovou náročnost až o 20 procent z původních 680ms na 550ms. Také jsme snížili počet cache misses, a to zejména v build fázi, kde rozdíl činí 10 procent.

7.4 Průběh GRACE hash-joinu

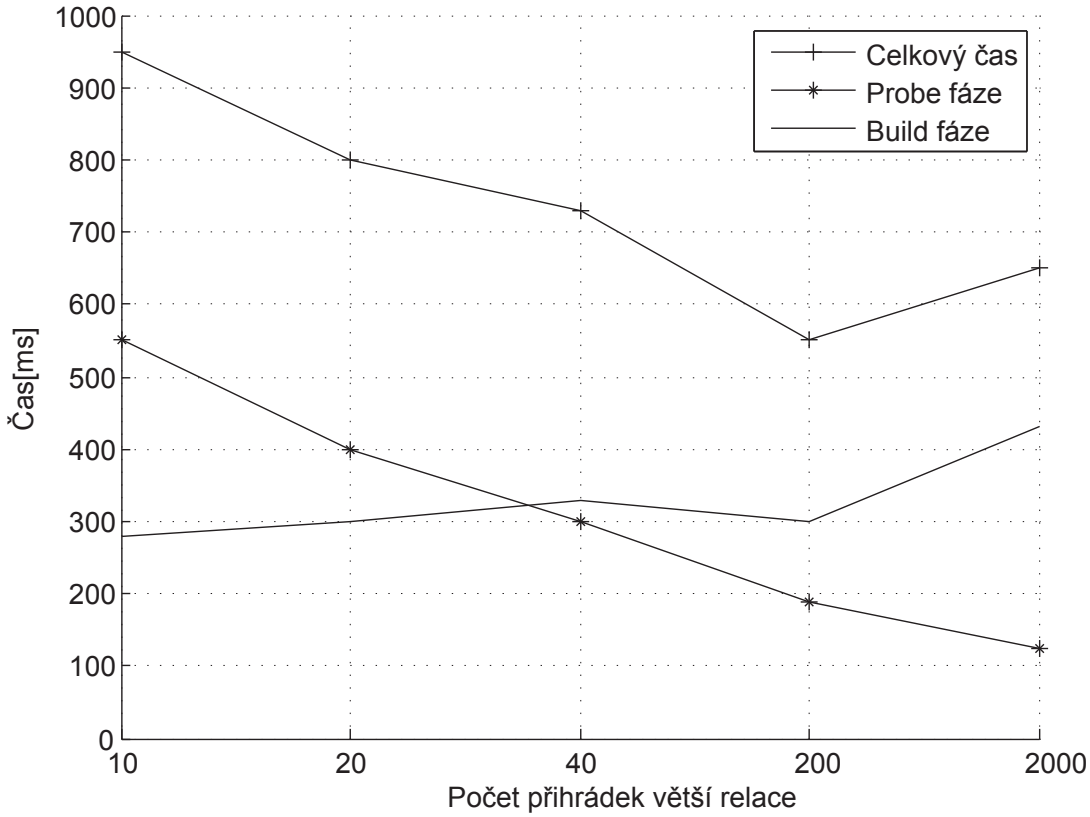
Testování probíhalo na PC2 a vstupní relace měly opět velikost jeden a pět miliónů. Pro měření byly konstantní velikosti všech atributů, kromě počtu přihrádek větší relace. Výsledná data můžeme vidět na grafu číslo 2, který se skládá ze tří datových řad, a to z celkového času, času probe fáze a času build fáze větší relace. Z grafu číslo 2 pak můžeme vyvodit několik zajímavých faktů.

- Čas probe fáze se snižuje s roustoucím počtem přihrádek.
- Čas build fáze roste s navyšováním počtu přihrádek.
- Celkový čas zvyšováním počtu přihrádek klesá do té doby, dokud benefit v probe fázi nezastíní zpomalení build fáze.

Časový nárust u build fáze je zapříčiněn tím, že se zvedá počet přihrádek a tedy i počet paměťových regionů, ze kterých je potřeba číst a zapisovat, a tedy je třeba držet v cache více referencí. Ve výsledku to pak znamená, že dojde ke cache misses a tedy i ke snížení výkonu.

Průběh grafu také dokazuje existenci L1 cache. Počet přihrádek o hodnotě 20 je hodnota optimalizovaná pro L2 cache a jelikož má L1 cache menší velikost, je zapotřebí větší množství přihrádek. V této konfiguraci je tento počet 312 a jak můžeme vidět, celkový čas se přibližně do této hodnoty rapidně snižuje. Výsledné počty cache misses pak tento graf podporují.

Průměrný počet cache misses build fáze vzrostl z 2,3 miliónu pro 20 přihrádek na 4 milióny pro 2000 přihrádek. U probe fáze pak počet cache misses klesl z 8 miliónů na 4 milióny.

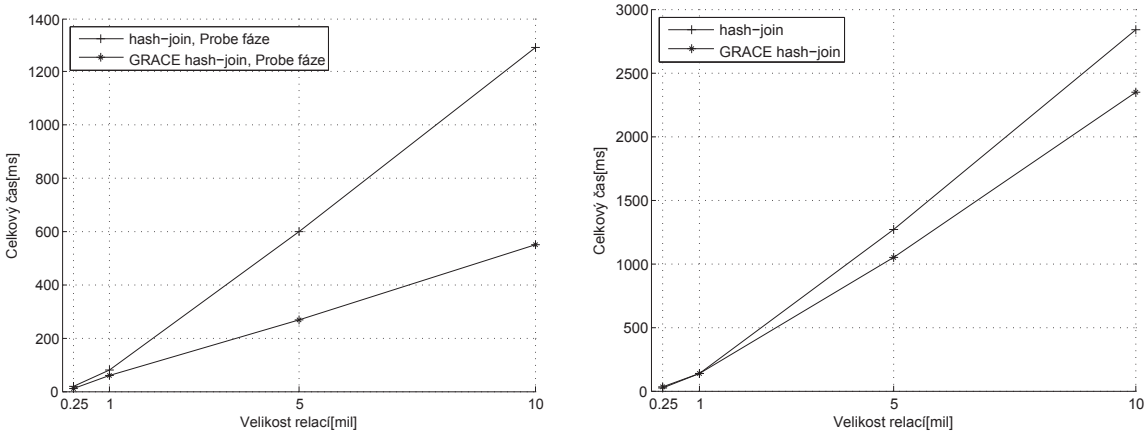


Graf 2: Průběh GRACE hash-joinu

7.5 Porovnání GRACE hash-joinu s hash-joinem

V tomto testu byl porovnáván výkon těchto algoritmů zafixováním všech hodnot, kromě velikosti relací, které zde měly shodný počet prvků.[6] Výsledky testů pak ukazují grafy číslo 3, 4, 5 a 6. Tabulka 2 pak ještě dále ukazuje, kolik milisekund bylo navíc u GRACE hash-joinu potřeba pro build fázi větší relace.

Poznámka 7.2 Přestože název práce napovídá implementaci pro L2 cache, všechny testy byly provedeny s optimalizací pro L1 cache, jelikož výsledky optimalizace pro L2 nebyly příliš vypovídající. V zásadě lze říci, že pro L2 cache měly výsledky shodný průběh, tedy zvýšení času v build fázi a snížení času v probe fázi. Rozdíl byl v tom, že toto snížení nebylo dostatečné, aby byl výsledný rozdíl markantnějšího charakteru, nebo byl nulový. V některých případech byl dokonce obyčejný hash-join výkonnější než GRACE hash-join, a to i při takových testech, kdy L1 cache optimalizovaný GRACE hash-join exceloval.

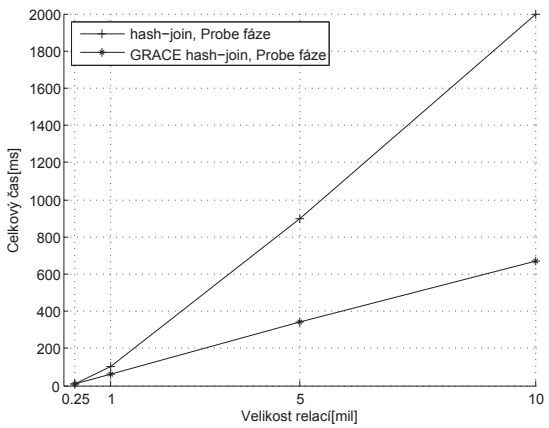


(a) Čas probe fáze

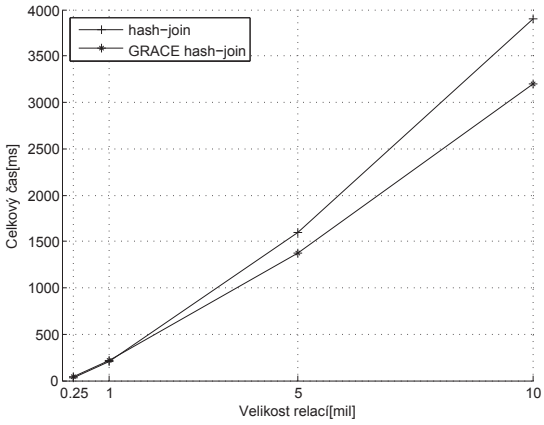
(b) Celkový čas

Graf 3: Porovnání GRACE hash-joinu s hash-joinem, PC1

[6] Shodný počet prvků byl zvolen, jelikož jsme se snažili napodobit testy z článku [1], kde byly také použity shodné velikosti relací.

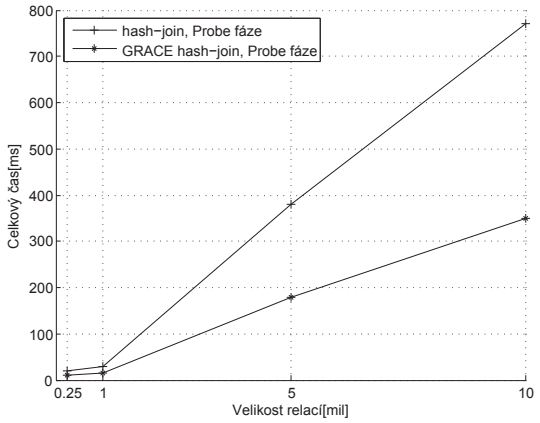


(a) Čas probe fáze

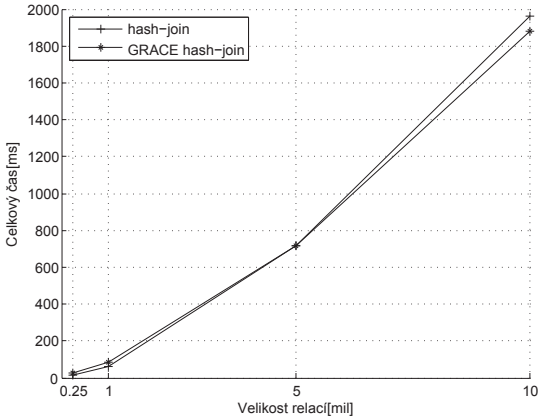


(b) Celkový čas

Graf 4: Porovnání GRACE hash-joinu s hash-joinem, PC2

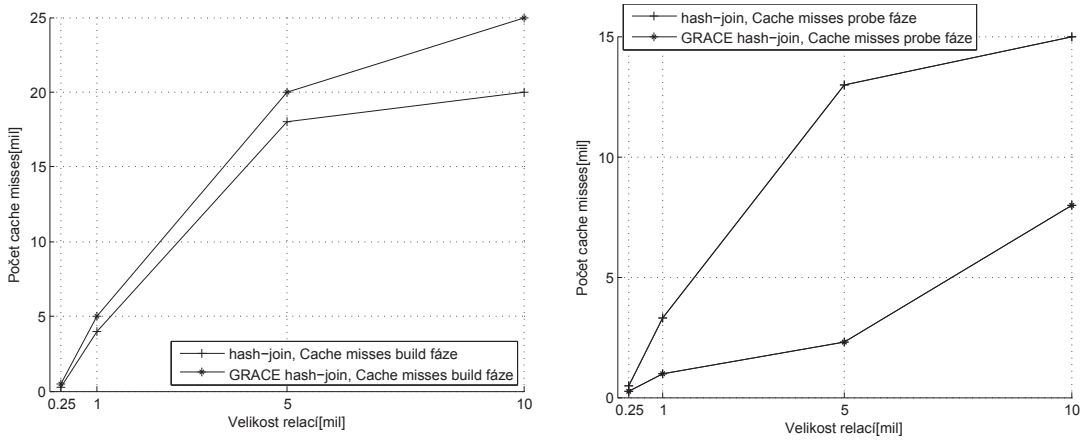


(a) Čas probe fáze



(b) Celkový čas

Graf 5: Porovnání GRACE hash-joinu s hash-joinem, PC3



(a) Počet cache misses build fáze

(b) Počet cache misses probe fáze

Graf 6: Porovnání cache misses GRACE hash-joinu s hash-joinem, PC2

Velikost relací [mil]	PC1	PC2	PC3
0.25	$5ms$	16ms	$4ms$
1	$25ms$	$40ms$	$15ms$
5	$110ms$	$300ms$	$100ms$
10	$300ms$	$780ms$	$300ms$

Tabulka 2: Časy build fází

Z grafů lze vyvodit, že až na PC3 došlo po optimalizaci k rapidnímu časovému zrychlení. Na tomto stroji (PC3) pak měly tyto optimalizce spíše negativní charakter. Při velikosti relací okolo deseti miliónů jsme ještě byli schopni pozorovat slabé zlepšení, nicméně se snižováním počtu prvků se stával GRACE hash-join pomalejší volbou. Přestože stejně jako u ostatních počítačů došlo ke snížení časů probe fáze, navýšená časová náročnost build fáze tento rozdíl eliminovala. Tyto výsledky lze přisuzovat tomu, že se nejedná o běžný počítač, ale o serverové PC. Je tedy pravděpodobné, že má nízké cache zpoždění a jeho paměť je velice rychlá.

U zbývajících dvou počítačů je pak průběh velice podobný. Pro velké vstupní relace dominuje GRACE hash-join, ale se snižováním počtu prvků dochází k úbytku výkonu a jakmile se dostaneme na určitou úroveň, začne být rychlejší hash-join.

Podle předpokladu skončilo i porovnání cache misses obou algoritmů. Zatímco probe fáze měla mnohem méně cache misses u GRACE hash-joinu, než u hash-joinu, build fáze optimalizovaného algoritmu zaznamenávala nárůst v počtu cache misses. [7]

[7] Počet cache misses je nutné brát jako orientační, jelikož výstup z AQTime se často liší pro jednotlivá spuštění pro stejné hodnoty. Opakovaní měření a zprůměrování hodnot však všechny zmíněné trendy potvrdilo.

Také můžeme vidět, že zrychlení probe fáze bylo znatelnější na PC číslo 2, kde byl čas snížen optimalizací až o dvě třetiny. Oproti tomu na PC číslo 2 docházelo ke zrychlení menšímu, a to o polovinu původních hodnot. Tento rozdíl přisuzujeme faktu, že PC číslo 1 má cache s kratšími zpožděními vyvolanými cache misses, a tak jejich eliminace nemá takový dopad. Jistý podíl zde bude mít i přítomnost Hyper-Threadingu u PC1.

7.6 Srovnání radix-cluster algoritmu s GRACE hash-join algoritmem

V poslední sekci se budeme věnovat testování radix-cluster hash-joinu. V případě, že pracujeme velice velkými relacemi, build fáze se stává úzkým místem celého algoritmu. Pro představu pokud máme relace o velikosti 25 miliónů prvků, trvá celý algoritmus na počítači číslo dva 9.5 sekundy, a z toho jen tvorba hashovací tabulky trvá 5 sekund. Tento fakt je způsoben tím, že zde máme enormní množství polí, do kterých můžeme hashovat. Přesněji je tento počet více než dva milióny. O omezení tohoto úzkého místa se snaží právě radix-cluster algoritmus.

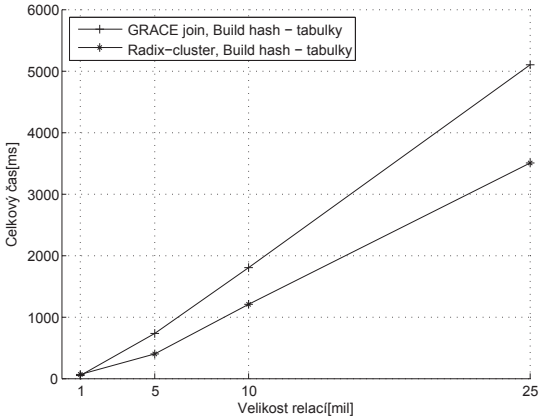
Během testování byla použita data jako u předchozího testu, tedy relace stejných velikostí, s prvky typu integer. Relace měly malou nerovnoměrnost uspořádání mezi jednotlivé přihrádky a také neexistoval prvek, který by se v případné výsledné relaci nevyskytoval.

U tohoto algoritmu je důležité správně určit, o kolik se každým průchodem zvětší počet polí v hashované tabulce. Jak již bylo řečeno v předchozí kapitole, abychom co nejvíce omezili počet cache misses, respektive TLB misses, měl by tento počet být menší, než je počet cache lines, respektive počet TLB entries.

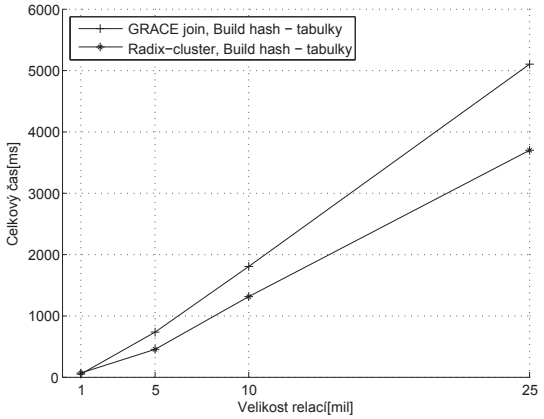
Všechny testy na všech počítačích přinesly srovnatelné výsledky. Při velikosti relací menší než jeden milión prvků měla optimalizace spíše negativní dopad, a všechny optimalizace se ukázaly jako nevhodné. Vyjímkou v tomto pravidle byl PC2, na kterém i pro tyto malé velikosti běžel radix-cluster hash-join s až 20 procentním zrychlením, než GRACE hash-join. Tento fakt můžeme opět usuzovat tomu, že PC2 má nejvyšší cenu za cache miss, a zmenšený počet misses se na něm projeví nejvíce.

Pro tento počítač jsme také měřili počet cache misses, které mnohonásobně klesaly z původních hodnot, což nám potvrdilo, že je tento algoritmus cache optimalizací.

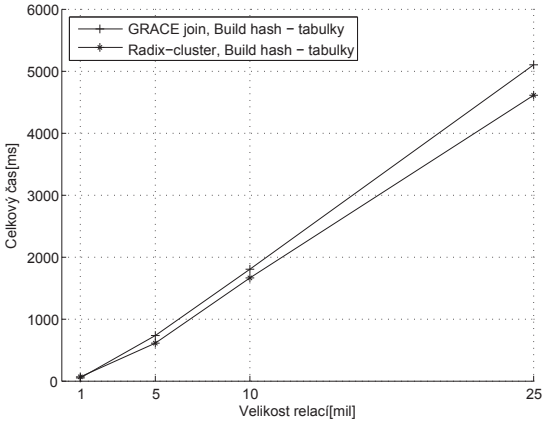
Grafy 7 popisují tento test pro PC1, grafy 8 pro PC2 a grafy 9 pro PC3. Grafy 10 pak ještě zobrazují počty cache misses pro PC2. Proměnná B značí, kolikrát se zvětšil počet hashovaných polí každým průchodem, tedy vstupní hodnotu algoritmu, která nám říká, kolik nových hashovaných polí vznikne z každého existujícího pole. Počet průchodů pak ukazuje tabulka 3.



(a) Časy build fází, B = 32



(b) Časy build fází, B = 128

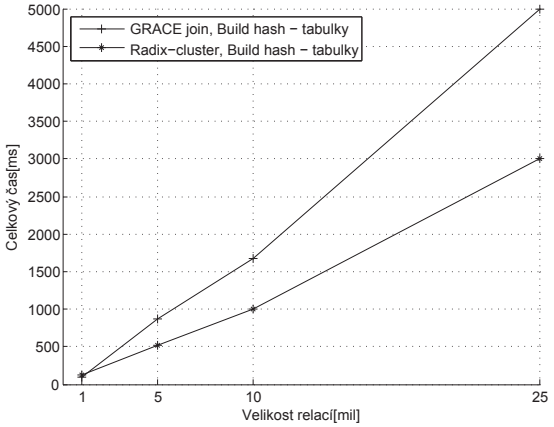


(c) Časy build fází, B = 1024

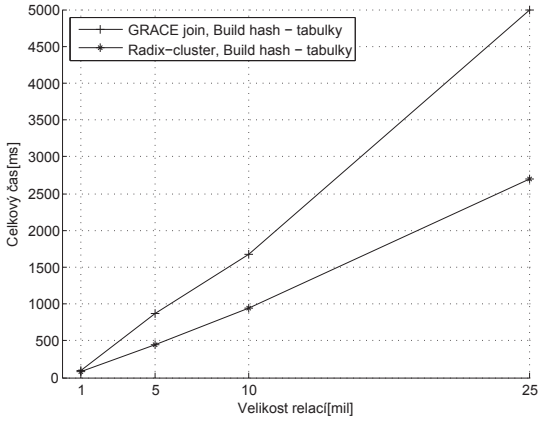
Graf 7: Porovnání GRACE hash-joinu s hash-joinem, PC1

Velikost relací [mil]	B = 32	B = 128	B = 1024
1	5	3	3
5	4	3	2
10	4	3	2
25	4	3	2

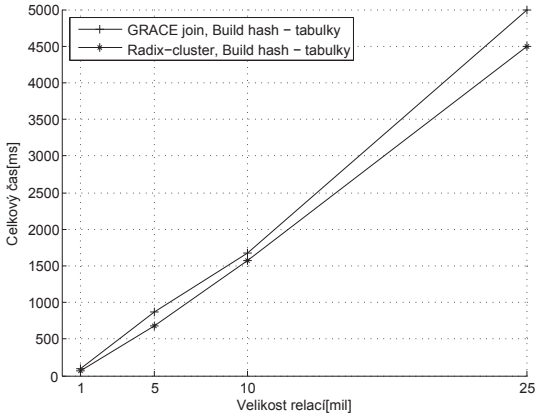
Tabulka 3: Počty průchodů



(a) Časy build fází, B = 32

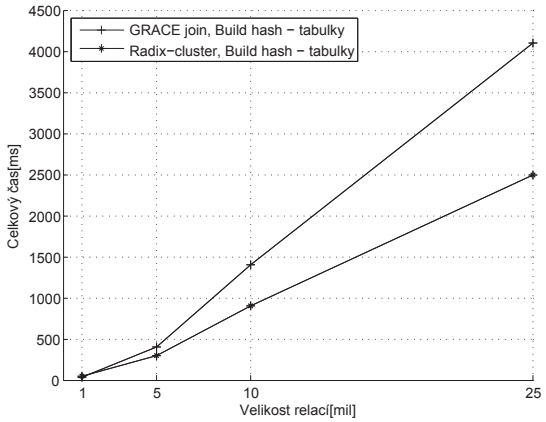


(b) Časy build fází, B = 128

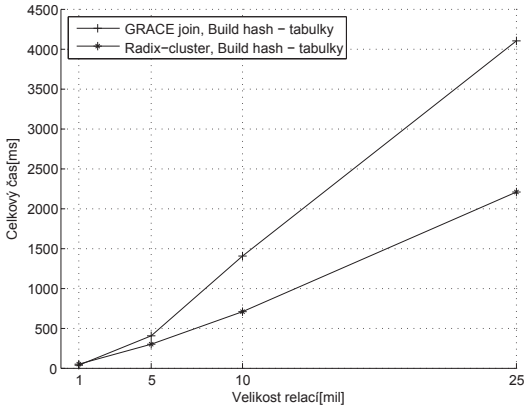


(c) Časy build fází, B = 1024

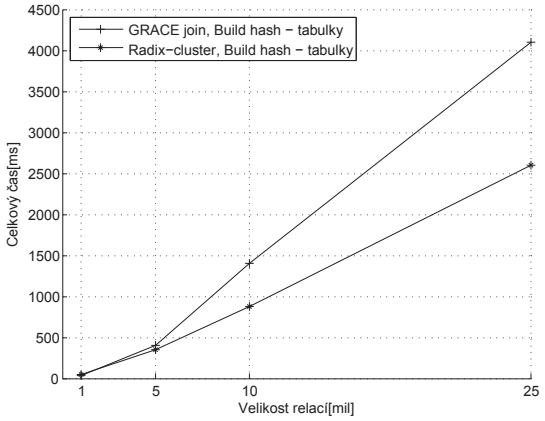
Graf 8: Porovnání GRACE hash-joinu s hash-joinem, PC2



(a) Časy build fází, B = 32

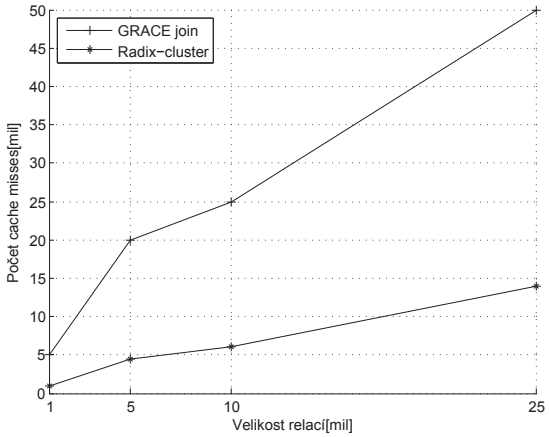


(b) Časy build fází, B = 128

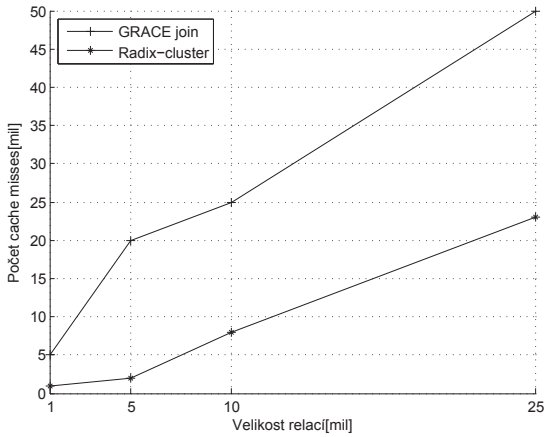


(c) Časy build fází, B = 1024

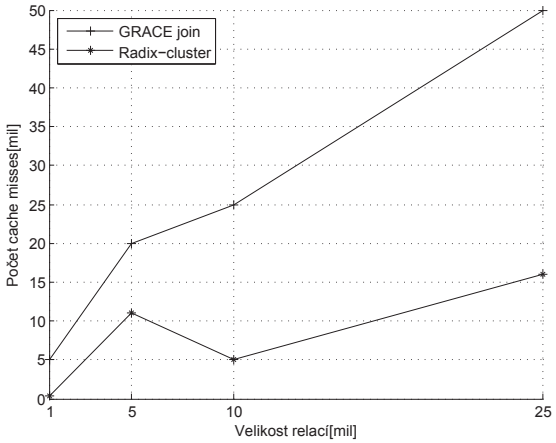
Graf 9: Porovnání GRACE hash-joinu s hash-joinem, PC3



(a) Počty cache misses build fáze, B = 32



(b) Počty cache misses build fáze, B = 128



(c) Počty cache misses build fáze, B = 1024

Graf 10: Porovnání cache misses GRACE hash-joinu s hash-joinem, PC2

8 Závěr

Cílem této práce bylo podat čtenáři ucelené informace o operaci relačního spojení, procesorové cache a zejména pak o tvorbě algoritmů, které jsou pro cache optimalizovány. Hlavní část práce se zabývala implemetací a následným testováním tří algoritmů. Radix-cluster hash-join, GRACE hash-join a poté také hash-join, který jakožto neoptimalizovaný algoritmus posloužil k vzájemnému testování.

Podrobné testování na třech rozdílných počítačích ukázalo, že procesorová cache má nezastupitelnou funkci v moderních počítačích a jejím využitím lze zrychlit algoritmy o desítky procent. Zjednodušeně můžeme říci, že jsme při zde implementovaných algoritmech vyměnili pomalý přístup do paměti za rychlé CPU operace. Tyto algoritmy nemusí být omezeny pouze na operaci relačního spojení, ale lze je použít i v ostatních odvětvích programování, zejména tam, kde se pracuje s velkým množstvím dat.

V závislosti na vstupních datech a typu CPU můžeme pozorovat u GRACE hash-joinu snížení doby běhu až o 30% oproti neoptimalizovanému hash-joinu. Radix-cluster hash-join pak v našich testech dosahoval dalšího zrychlení, a to až o 40%. Obě optimalizace, zejména však radix-cluster hash-join v porovnání s GRACE hash-joinem také přinesly rapidní pokles v počtu cache misses. Nejlepší zrychlení jsme pozorovali na PC číslo 2. Tento fakt usuzujeme tomu, že má tento procesor největší cenu za cache miss, nemá L3 cache a také nedisponuje technologiemi jako je například Hyper-Threading.

Testy také ukázaly, že je důležité správně zvolit korektní algoritmus pro dané vstupy a daný počítač, jelikož i optimalizovaný, neoptimálně zvolený algoritmus bude pomalejší než neoptimalizovaný.

Práce dále potvrdila poznatky zmíněné v článku[1] z roku 2002, kde byly porovnávány stejné algoritmy, a kde byly vyvozeny závěry, že přístup do paměti je úzkým místem celého systému, a že se tento trend v následujících letech nezmění. Je zajímavé, že i po téměř 15 letech rychlého vývoje počítačů jsou tyto poznatky stále velice aktuální. Zda-li bude tento trend pokračovat nadále je otázkou, jelikož například testovací procesor číslo 3 již při některých testech tyto výsledky nevykazoval. Nicméně lze usoudit, že v blízké době si bude cache stále nacházet své místo mezi vhodnými kandidáty na optimalizaci.

Z pohledu dalšího vývoje projektu lze algoritmy dále optimalizovat použitím pokročilého nízkoúrovňového programování a naprogramování dalších hashovacích funkcí pro jednotlivé datové typy. V neposlední řadě by bylo vhodné algoritmy hlouběji analyzovat, abychom získali jasná pravidla výběru nejlepšího algoritmu pro všechny možné vstupy.

9 Reference

[1] Manegold, Boncz, Kersten *Optimizing Main-Memory Join on Modern Hardware, Knowledge and Data Engineering, IEEE Transactions on*,Díl: 14, Vydání: 4, strany: 709-730, [online] Dostupné z:

`http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=1019210`

[2] Thomas Zurek *Sort-Merge Joins*, [online] Dostupné z:

`http://www.dcs.ed.ac.uk/home/tz/phd/thesis/node20.htm`

[3] Craig Freedman *Hash Join*, [online] Dostupné z:

`http://blogs.msdn.com/b/craigfr/archive/2006/08/10/687630.aspx`

[4] Craig Freedman *Merge Join*, [online] Dostupné z:

`http://blogs.msdn.com/b/craigfr/archive/2006/08/03/687584.aspx`

[5] Petr Veselík, Časopis Computer *Vyznejte se v procesoru – velký přehled technologií*, [online] Dostupné z:

`http://www.zive.cz/clanky/vyznejte-se-v-procesoru--velky-prehled-technologi sc-3-a-147124/`

[6] *H*ENNESSY, John L a David A PATTERSON. Computer architecture: a quantitative approach. 5th ed. Waltham, MA: Morgan Kaufmann/Elsevier, c2012, 289. ISBN 9780123838728.

[7] Markus Kowarschik, Christian Weis *An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms*, [online] Dostupné z:

`http://www.cc.gatech.edu/~bader/COURSES/UNM/ece637-Fall2003/papers/KW03.pdf`

[8] PC Tech Guide *What is L2 (Level 2) cache memory?*, [online] Dostupné z:

`http://www.pctechguide.com/computer-memory/what-is-l2-level-2-cache-memory`

[9] Bob Brown *Cache Memory Mini-Lecture*, [online] Dostupné z:

`http://bbrown.spsu.edu/web_lectures/cache/`

[10] *An Overview of Cache*, [online] Dostupné z:

`http://download.intel.com/design/intarch/papers/cache6.pdf`

[11] Ruye Wang *Cache Memory: Replacement Policy*, [online] Dostupné z:

`http://fourier.eng.hmc.edu/e85_old/lectures/memory/node5.html`

[12] Ulrich Drepper *Memory part 2: CPU caches*, [online] Dostupné z:
http://lwn.net/Articles/252125/

[13] Mark Hill *Mark D. Hill*, [online] Dostupné z:
http://pages.cs.wisc.edu/~markhill/

[14] Shaaban *Types of Cache Misses: The Three C's*, [online] Dostupné z:
http://meseec.ce.rit.edu/eecc551-winter2001/551-1-30-2002.pdf

[15] Wikipedia *CPU cache*, [online] Dostupné z:
http://en.wikipedia.org/wiki/CPU_cache

[16] Shannon Cepeda (Intel) *What You Need to Know About Prefetching*, [online] Dostupné z:
https://software.intel.com/en-us/blogs/2009/08/24/what-you-need-to-know-about-prefetching

[17] Agner Fog *Agner's CPU blog*, [online] Dostupné z:
http://www.agner.org/optimize/blog/read.php?i=6

[18] ARPACI-DUSSEAU *Paging: Faster Translations (TLBs)*, [online] Dostupné z:
http://pages.cs.wisc.edu/~remzi/OSTEP/vm-tlbs.pdf

[19] Joel Hruska *How L1 and L2 CPU caches work, and why they're an essential part of modern chips*, [online] Dostupné z:
http://www.extremetech.com/extreme/188776-how-l1-and-l2-cpu-caches-work-and-2

[20] Manegold *The Calibrator (v0.9e), a Cache-Memory and TLB Calibration Tool*, [online] Dostupné z:
http://homepages.cwi.nl/~manegold/Calibrator/

[21] SmartBear *Application Performance - AQTime Pro*, [online] Dostupné z:
http://smartbear.com/products/qa-tools/application-performance-profiling/

[22] SmartBear Support *Running windows in debug mode*, [online] Dostupné z:
http://support.smartbear.com/viewarticle/8720/

[23] Cozzini, Stefano, Democritos/ICTP course in "Tools for computational physics
Optimization techniques: an overview.., [online] Dostupné z:
http://www.democritos.it/events/computational_physics/lecture_stefano3.pdf

[24] Boost C++ LIBRARIES *Boost C++ Libraries*, [online] Dostupné z:
http://www.boost.org/

[25] cplusplus.com *Macro RAND_MAX*, [online] Dostupné z:
http://www.cplusplus.com/reference/cstdlib/RAND_MAX/

[26] SmartBear Support *Supported Processor Models*, [online] Dostupné z:
http://support.smartbear.com/viewarticle/43584/